**Ex-1**

**Why Data Structures and Algorithms are Essential:**

* **Efficiency:** Handling large inventories involves managing numerous items and operations (add, update, delete, query). Efficient data structures and algorithms help ensure these operations are performed quickly and use minimal resources.
* **Scalability:** As the number of products increases, It helps to handle data efficiently

**Suitable Data Structures:**

* **ArrayList:** Suitable for scenarios where the order of products matters and you frequently access items by index. However, operations like insertion and deletion can be costly due to shifting elements.
* **HashMap:** Ideal for scenarios where fast lookups, additions, and deletions are needed. Each product can be accessed quickly via its productId, which serves as a key.

I have used hash Map in my code because it has key value pairs and easy to acess elements in hash map than array list

**Time Complexity Analysis:**

* **Add Product**: O(1) - Inserting into a HashMap is an average-case constant time operation.
* **Update Product**: O(1) - Retrieving and updating a product by its ID in a HashMap is an average-case constant time operation.
* **Delete Product**: O(1) - Deleting an entry from a HashMap is an average-case constant time operation.

**Ex-2**

**Understanding Asymptotic Notation**

**Big O Notation**: Big O notation is a mathematical representation used to describe the efficiency of an algorithm, particularly in terms of time complexity and space complexity. It provides an upper bound on the growth rate of an algorithm’s runtime or space requirement relative to the input size. This notation helps in comparing the performance of different algorithms, especially when dealing with large inputs.

**How It Helps in Analyzing Algorithms**:

* **Scalability**: Big O notation helps determine how well an algorithm scales with increasing input size
* **Worst-Case Scenario**: It gives a way to understand the worst-case performance, which is crucial for guaranteeing that the algorithm performs within acceptable limits for all possible inputs.
* **Comparative Analysis**: By providing a high-level abstraction, Big O notation allows you to compare different algorithms and choose the one that performs better with large datasets.

**Big O Notation Examples:**

* **O(1)**: Constant time complexity. The runtime does not change with the size of the input (e.g., accessing an element in an array).
* **O(n)**: Linear time complexity. The runtime grows linearly with the input size (e.g., iterating through an array).
* **O(log n)**: Logarithmic time complexity. The runtime grows logarithmically with the input size (e.g., binary search).
* **O(n^2)**: Quadratic time complexity. The runtime grows quadratically with the input size (e.g., bubble sort).

**Best, Average, and Worst-Case Scenarios for Search Operations**

**1. Linear Search**:

* **Best Case**: O(1) – The target element is the first element of the list.
* **Average Case**: O(n) – The target element is somewhere in the middle of the list, requiring on average half of the list to be checked.
* **Worst Case**: O(n) – The target element is the last element or not present in the list, requiring all elements to be checked.

**2. Binary Search**:

* **Best Case**: O(1) – The target element is at the middle of the list.
* **Average Case**: O(log n) – The search process requires repeatedly halving the search space, leading to a logarithmic time complexity.
* **Worst Case**: O(log n) – The search continues until the subarray is reduced to one element, or the target element is not found.

**Analysis**

**Comparing Time Complexity of Linear and Binary Search Algorithms**

* **Linear Search**:
  + Time Complexity: O(n)
  + Space Complexity: O(1)
  + Suitable for small or unsorted lists, where you need to check each element one by one.
* **Binary Search**:
  + Time Complexity: O(log n)
  + Space Complexity: O(1) for iterative implementation; O(log n) for recursive implementation (due to call stack)
  + Requires the list to be sorted before performing the search.

**Which Algorithm is More Suitable?**

* **For Small or Unsorted Lists**: Linear search is suitable because it does not require the list to be sorted and is simple to implement.
* **For Large or Sorted Lists**: Binary search is more efficient for large datasets where the list is already sorted, as its logarithmic time complexity significantly reduces the number of comparisons needed

**Ex-3**

**Understanding Algorithms**

**1. Bubble Sort**:

**Definition:**

Compare adjacent elements and swap them if they are in the wrong order. Repeat this process until the list is sorted.

**Time Complexity**:

* + **Best Case**: O(n) with optimization.
  + **Average Case**: O(n²).
  + **Worst Case**: O(n²).

**2. Insertion Sort**:

**Definition**:

Build a sorted portion of the list one element at a time. Take each new element and insert it into its correct position within the sorted portion.

**Time Complexity**:

* + **Best Case**: O(n) – When the list is already sorted.
  + **Average Case**: O(n²) – When the list is unsorted.
  + **Worst Case**: O(n²) – When the list is in reverse order.

**3. Quick Sort**:

**Definition**:

Select a pivot element from the list, partition the other elements into two sub-arrays according to whether they are less than or greater than the pivot. Recursively apply the same process to the sub-arrays.

**Time Complexity**:

* + **Best Case**: O(n log n).
  + **Average Case**: O(n log n).
  + **Worst Case**: O(n²).

**4. Merge Sort**:

**Definition**:

Divide the list into halves, sort each half recursively, and then merge the sorted halves back together.

**Time Complexity**:

* + **Best Case**: O(n log n).
  + **Average Case**: O(n log n).
  + **Worst Case**: O(n log n)

**Analysis**

**Bubble Sort**:

* **Time Complexity**:
  + **Best Case**: O(n) – When the list is already sorted, and we use an optimized version that detects no swaps are needed.
  + **Average Case**: O(n²) – When the list is unsorted, and it has to perform many swaps and comparisons.
  + **Worst Case**: O(n²) – When the list is sorted in reverse order.

**Quick Sort**:

* **Time Complexity**:
  + **Best Case**: O(n log n) – When the pivot divides the list into two nearly equal halves.
  + **Average Case**: O(n log n) – Generally, Quick Sort performs well on average.
  + **Worst Case**: O(n²) – When the pivot is the smallest or largest element, leading to highly unbalanced partitions (e.g., if the list is already sorted).

**Why Quick Sort is Generally Preferred Over Bubble Sort**

* **Efficiency**:
  + **Quick Sort** is much faster on average due to its O(n log n) time complexity compared to Bubble Sort’s O(n²). This makes Quick Sort suitable for larger datasets.
  + **Bubble Sort** is simple but inefficient for large lists because its time complexity grows quadratically with the size of the input.
* **Performance**:
  + Quick Sort generally outperforms Bubble Sort because it reduces the problem size more rapidly by dividing the list into smaller partitions.

**Ex-4**

**1. Understanding Array Representation:**

Array Representation in Memory

* **Contiguous Block:** Arrays are stored in a single, continuous block of memory. Each element is located next to the previous one.
* **Index-Based Access:** Elements are accessed using indices. The index tells you how far to move from the start of the array to get to the desired element.

Advantages of Arrays

* **Fast Access:** Accessing any element by its index is very fast (constant time, O(1)).
* **Efficient Memory Use:** Arrays use memory efficiently because they don’t require extra space
* **Simple to Use:** Easy to implement and use, with predictable performance.

**2. Time complexity and limitations**

* Add Operation

Time Complexity:

O(1): Adding an element to an ArrayList typically takes constant time, O(1), because ArrayList can grow dynamically and manages capacity internally.

* Search Operation

Time Complexity:

O(n): Searching for an employee involves iterating through the ArrayList to find the employee with the matching ID. In the worst case,we check every element in the ArrayList, resulting in a linear time complexity, O(n), where n is the number of employees.

* Traverse Operation

Time Complexity:

O(n): Traversing the entire ArrayList requires visiting each element once to display it.

* Delete Operation

Time Complexity:

O(n): Similar to the search operation, deleting an employee involves finding the employee by iterating through the ArrayList (O(n)) and then removing it

Limitations of Arrays

* Resizing Costs:
  + Although adding elements to an ArrayList is typically O(1) on average, resizing (when the internal array's capacity is exceeded) involves creating a new, larger array and copying existing elements to it. This resizing operation is O(n), where n is the number of elements in the ArrayList. However, resizing happens infrequently, so the average cost is spread out.
* Insertion/Deletion Costs:
  + O(n) Complexity: Inserting or deleting elements, especially in the middle of the list, requires shifting elements to maintain contiguous storage. This operation has a time complexity of O(n), where n is the number of elements to shift.
* Capacity Management:
  + Unused Space: ArrayList often allocates more space than needed to accommodate future growth, which can lead to unused capacity. This can lead to wasted memory, particularly if the array grows significantly larger than the number of elements it holds.
* Additional Memory for Internal Structures:
  + ArrayList uses an internal array to store its elements, which means that memory is used for both the internal array and any elements stored in it. This can be inefficient if the list is very sparse or if the internal array is resized frequently.
* Index-Based Access:

While accessing elements by index is very efficient (O(1)), other operations like searching for an element (if not by index) or complex queries may not be as efficient and could require iterating through the list.

**Ex-5**

**Types of Linked Lists**

**1. Singly Linked List**

* **Structure**:
  + Each node in a singly linked list contains two parts:
    - **Data**: The value stored in the node.
    - **Next**: A reference (or pointer) to the next node in the sequence.
  + The list starts with a head node and ends with a node whose next reference is null.
* **Operations**:
  + Insertion Search
  + Deletion Traversal

**2. Doubly Linked List**

* **Structure**:
  + Each node contains three parts:
    - **Data**: The value stored in the node.
    - **Next**: A reference to the next node in the sequence.
    - **Prev**: A reference to the previous node in the sequence.
  + The list has a head and often a tail
* **Operations**:
  + Insertion Search
  + Deletion Traversal

**Time Complexity of Linked Lists**

**Singly Linked List**

* Insertion at the Head: O(1)
* Insertion at the End: O(n)
* Insertion at a Specific Position: O(n)
* Deletion: O(n)
* Search: O(n)
* Traversal: O(n)

**Doubly Linked List**

* Insertion at the Head: O(1)
* Insertion at the End: O(1)
* Insertion at a Specific Position: O(n)
* Deletion: O(n)
* Traversal: O(n)

**Advantages of Linked Lists Over Arrays**

1. **Dynamic Size**:
   * **Linked Lists**: Can grow or shrink dynamically as nodes are added or removed. There’s no need to allocate a large block of memory upfront or handle resizing operations.
   * **Arrays**: Have a fixed size defined at creation. Changing the size requires creating a new array and copying elements, which can be costly.
2. **Efficient Insertions and Deletions**:
   * **Linked Lists**: Allow for efficient insertions and deletions at various positions, especially if you already have a reference to the position. This is because nodes can be added or removed without shifting other elements.
   * **Arrays**: Inserting or deleting elements, particularly in the middle of the array, requires shifting elements to maintain order, which can be inefficient.
3. **Flexible Memory Usage**:
   * **Linked Lists**: Use memory dynamically, allocating space for each node as needed. This avoids issues with memory fragmentation and unused space.
   * **Arrays**: Require a contiguous block of memory. If the array is resized, it can lead to inefficient use of memory and require time-consuming copying.
4. **No Need for Resizing**:
   * **Linked Lists**: Automatically adjust their size as nodes are added or removed, with no need for manual resizing or reallocation.
   * **Arrays**: Resizing involves creating a new array and copying elements, which can be a time-consuming operation.

**Ex-6**

**1. Understanding Search Algorithm**

**Linear Search**

Linear search, also known as sequential search, is a simple search algorithm that checks each element in a list or array one by one until the target element is found or the end of the list is reached.

**Example:** Suppose you have an array [3, 5, 7, 9, 11] and want to find the value 9.

* Start at the beginning: check 3 (not a match), then 5 (not a match), then 7 (not a match), then 9 (match found).

**Time Complexity:**

* **Best Case:** O(1) - The target is at the first position.
* **Average Case:** O(n) - On average, you might need to check half the elements.
* **Worst Case:** O(n) - The target is not present or is at the last position.

**Binary Search**

Binary search is an efficient search algorithm that works on sorted arrays or lists. It repeatedly divides the search interval in half, checking if the target is in the left or right half of the current interval.

**Example:** Suppose you have a sorted array [3, 5, 7, 9, 11] and want to find the value 9.

* Check the middle element (7). Since 9 > 7, search the right half [9, 11].
* Check the middle element of the right half (9). Match found.

**Time Complexity:**

* **Best Case:** O(1) - The target is at the middle of the array.
* **Average Case:** O(log n) - Each comparison halves the search space.
* **Worst Case:** O(log n) - The search space is halved each time until the target is found or the space is exhausted.

**Comparison of Time Complexity**

* **Linear Search:**
  + **Best Case:** O(1)
  + **Average Case:** O(n)
  + **Worst Case:** O(n)
* **Binary Search:**
  + **Best Case:** O(1)
  + **Average Case:** O(log n)
  + **Worst Case:** O(log n)

**When to Use Each Algorithm**

* **Linear Search:**
  + Use when the list is unsorted or when the data set is small.
  + Use when simplicity is preferred and sorting is not feasible.
* **Binary Search:**
  + Use when the list is sorted and the data set is large.
  + Preferred for efficient searching in large data sets due to its logarithmic time complexity.

**Ex-7**

**Understand Recursive Algorithms:**

Recursion is when a function solves a problem by calling itself. It keeps breaking the problem into smaller pieces until it reaches a simple case it can handle directly. This can make complex problems easier to solve by handling them in smaller, more manageable parts.

**Time Complexity of Recursive Algorithms**

The time complexity depends on how many times the function calls itself. Simple cases might be O(n)O(n)O(n), but complex ones can be exponential, like O(2n)O(2^n)O(2n).

**Optimizing Recursion**

To avoid excessive computation, use **memoization** to store results of previous calls or **dynamic programming** to systematically build solutions, reducing redundant calculations.